Amendments to the Specification:

Please replace the paragraph beginning on page 1, line 14 with the following amended paragraph:

With the field of compilers, the problem of handling exceptions is important for languages that permit them. For example, the C++ and ~~Java~~ JAVA programming language permit a program to "throw an exception", which means that a structure (the exception) is constructed, and then transmitted (thrown) up the execution stack until a handler catches the exception, at which point normal execution continues. A key feature of both C++ and ~~Java~~ JAVA programming language is that they permit the programmer to specify cleanup actions to be performed if an exception is thrown. In C++, these actions are destructions of objects that are popped from the execution stack by the throwing of the exception. In JAVA programming language ~~Java~~, these actions are code specified by "finally" clauses.

Please replace the paragraph beginning on page 3, line 14 with the following amended paragraph:

For the destruction of an item, the tag is DESTROY. Field 203 "object" points to the object to be destroyed, and field 204 "dtor" points to the code to invoke for that item. For other sorts of cleanup, say the ~~Java~~ JAVA programming language "finally" clause, or C++ "exception specifications", the field "dtor" points to the code to be executed, and the "object" field is not used, or points to some sort of data structure that further specifies the cleanup to be done. (The exact nature of the

cleanup is beyond the scope of this disclosure, and therefore the cleanups in the example are restricted to simple destructions.)

Please replace the paragraph beginning on page 8, line 6 with the following amended paragraph:

FIG. 4 shows an overview of the steps of a method in accordance with the present invention. The steps permits a much more general optimization than just for exception handling, and thus are stated in a general form. The general problem is that given a set of selected program points, insert operations into the program that guarantee that the data structure will be in the correct state at the selected points (but perhaps in an incomplete or incorrect state at other points). The selection of program points depends upon the application of the invention. For example, when applying it to optimizing the EH stack, the selected points are those just before an instruction that might throw an exception. Step 400 determines the state of the data structure at the selected program points. There are a multitude of ways to do this determination, including dataflow analysis and abstract interpretation. For the exception-handling example, the data structure is the EH stack. Step 401 partitions the state of the data structure into components. In our example, the components are [[and]] the EH_stack_ptr and the fields of the EH_item. Step 402 determines operations to be inserted at each selected program point in order to set each component of the state at said program point. In our example, these operations are the assignments and call to setjmp required to set EH_stack_ptr and the fields of each EH. The goal of steps 400-402 is to convert a problem that might be difficult

or impossible for partial-redundancy elimination into a problem that it can handle easily. Steps 403 and 404 perform the analysis and transformation respectively that eliminate partial redundancies.

Please replace the paragraph beginning on page 8, line 27 with the following amended paragraph:

FIG. 5 shows a control-flow graph representation of the program in FIG. 1. The graph is an edge-labeled graph, meaning that instructions are attached to edges, rather than vertices. An edge-labeled graph simplifies implementation and explanation of the present invention, but is not required. The present invention works equally well with other forms, such as the traditional vertex-labeled control-flow graph. Each edge, before transformation, has either a single instruction or a label indicating a branch decision. After transformation, there may be multiple instructions on an edge. Edges labeled "(true)" or "(false)" are taken if the preceding instruction evaluates to true [[of]] or false respectively. Edges labeled "(fail)" are taken if the preceding instruction throws an exception. Edges labeled "(succeed)" are taken if the preceding instruction does not throw an exception. Edges denoted by dashed lines denote "exceptional" execution: these are edges that ultimately will be removed by methods in accordance with the present invention. Edges that are not "exceptional" are called "normal".

Please replace the paragraph beginning on page 11, line 22 with the following amended paragraph:

FIG. 7 shows the required EH stack states for the example in FIG.1. For convenience, the reference numerals for vertices in FIG. 7 are exactly 200 more than those for the corresponding vertices in FIG. 5. Only normal edges are shown; exceptional edges have been removed. Valid EH stack states are required for each vertex that is the tail of an edge which has an operation that might throw an exception, because if an exception is thrown, further processing depends upon the EH stack. Those vertices are illustrated as small open circles. At all other vertices, the EH stack state is irrelevant, and does not have to be valid. The stack states shown are for cleaning up object declared in the program fragment of FIG. 1. The stack implicitly continues further down for objects outside the scope of said fragment; this implicit portion is called the "continuation stack". Thus the EH stack states for vertices 700 and 706 are simply the continuation stack, and must be valid at those vertices because respectively edges 700-701 and 706-726 might throw an exception. The EH stack state for vertices 708 and 712 specifies that (if an exception is thrown by edges 708-709 and 712 –713 respectively), object "cat" should be destroyed, the catch clauses for the try block inspected, and, if the exception is not caught, then object "ant" should be destroyed, followed by whatever the continuation stack specifies. The EH stack state for vertex 710 adds the destruction of object "dog" on top of the state for vertices 708 and 712, and is required because the [[cal]] call to "woof" on edge 710-711 might throw an exception. The EH stack state for vertex 720 specifies that if the call to woof () on edge 720-721 throws an exception, both objects "elk" and "ant" should be destroyed (followed by whatever the continuation stack specifies).

Please replace the paragraph beginning on page 13, line 18 with the following amended paragraph:

FIG. 9 shows method CLEANUP_TREE (u), which computes the portion of the cleanup tree required at a control-flow graph vertex u, and sets STATE[u] to the cleanup tree node representing an EH stack state. Step 900 checks whether STATE[u] has been set yet, and if so, there is no work to do. Otherwise step 901 sets STATE[u] to a special marker nil, to mark the fact that u is being processed. The marker keeps the method from infinitely recursing on control-flow graphs containing loops. Step 901 also initializes set E to the set of edges with tail vertex u. If set E is empty, then step 902 checks if u is the terminal or throw vertex, and if so, step 903 allocates the root node of the cleanup tree, and sets the maps appropriately. Step 903 allocates a node directly, instead of calling method ALLOCATE, because the root node needs special treatment. If set E is non-empty then step 904 chooses any edge in E for inspection. If it is not an exceptional edge, then step 905 checks whether e can thrown an exception. If so, step 906 adds the tail of e to set NEED. If in step 904 edge e is exceptional, then step 907 checks if e is the entry into a chain of catch clauses for a try block. If so, then step 908 invokes method CATCH_CHAIN vertex for u (which is the tail of e). Otherwise step 909 recursively invokes method CLEANUP_TREE on the head of e. The point of recursion is to traverse the rest of the exceptional edges the contribute to the EH stack state for vertex u. Step 910 checks if edge e destroys an object, and if not, step 911 simply propagates the state of its head back to its tail. If edge e in step 910 does destroy an object, then step 912

creates a cleanup tree node that represents the destruction. Step 913 removes *e* from *E* and repeats step 904 if there are more edges remaining.

Please replace the paragraph beginning on page 19, line 8 with the following amended paragraph:

FIG. 19 and 20 show a method for using the solutions UPSAFE* and DOWNSAFE* to place instructions that set components of EH stack.[[.]] FIG. 19 finds the edges for said insertion, and FIG. 20 determines what instructions to place. Step 1900 initializes set *E* to the set of edges. Steps 1901-1903 loop over each edge *e* in *E*. Steps 1904-1906 loop over each index *k* of the solutions to the flow equations. Steps 1907-1910 check whether the following conditions both apply to edge *e*:

- UPSAFE* (TAIL(*e*)) V DOWNSAFE* (TAIL(*e*)) is false.
- UPSAFE * (HEAD(*e*)) V DOWNSAFE* (HEAD(*e*)) is true.

If both conditions apply, step 1911 invokes method BIT_DIFFERENCE(*k*).

Please replace the paragraph beginning on page 19, line 18 with the following amended paragraph:

FIG. 20 shows method B BIT_DIFFERENCE(k). Throughout FIG. 20, the notation "Insert" means to insert an instruction on the edge e being processed in FIG. 19 when it invoked BIT_DIFFERENCE(k). The notation REC(n) means the EH_item associated with node n. There is a unique EH_item associated with each node n of the cleanup tree, except for the CONTINUE node, which has no corresponding EH_item because in effect its EH_item is whatever was on top of the any other instructions on the edge. Multiple instructions on an edge are permitted at this point in the transformation process. Step 2000 determines to which partial map (OBJECT_INDEX, SETJMP_INDEX, PTR_INDEX, OR KIND_INDEX) index k belongs. As noted earlier, no integer belongs to more than one of those maps. If k=OBJECT_INDEX(n) for some cleanup tree node n, then the corresponding flow equations deal with placement of a destruction operation. Step 2001 places an instruction that sets the "desctructor.object" field of the associated EH_item. If k=SETJMP_INDEX(n) for some cleanup tree node n, then step 2002 inserts a call to "setjmp". If k=PTR_INDEX(n) for some cleanup tree node n, then step 2003 determines whether n is the CONTINUE node. If so, then step 2004 inserts an assignment that sets EH_stack_ptr to point to SAVER.next, where SAVER is the EH_item in which prologue code will save (in SAVER.next) the EH_stack_ptr from the calling routine. If in step 2003, n is not the CONTINUE node, then step 2005 inserts an assignment that sets EH_stack_ptr to point to the EH_item associated with node n. If in step 2000, k=KIND_INDEX(n) for some cleanup tree node n, then step

2006 inspects the operation represented by *n*. If it is for a try-block, then step 2007 inserts an assignment to set the field "tag" of the associated EH_item to "TRY", and step 2008 inserts an assignment to se the field "try_block.catch_info". Otherwise, if in step 2005 the operation is not for a try-block, it must be for a destruction. Step 2009 sets the field "tag" of the associated EH_item to "DESTROY", and step 2010 inserts and assignment that sets the filed "destructor.dtor" to point to the destructor for the object to be destroyed. Steps 2011-2015 concern setting the field "next" of the associated EH_item. Step 2011 checks whether node *n* has a parent. If so, step 2012 checks wither this parent is the CONTINUE node. If not, step 2013 inserts an assignment that sets the filed "next" of the associated EH_item to point to the EH_item associated with the parent. If the parent in step 2012 is the CONTINUE node, then step 2014 checks whether *n* is the SAVER node. If so, the prologue will handle setting the "next" field. Otherwise, step 2014 inserts an assignment that sets the "next" field from the "next" field of the SAVER node.

Please replace the paragraph beginning on page 22, line 11 with the following amended paragraph:

The present invention's partial-redundancy elimination is a departure from standard practice of a partial redundancy elimination, because in effect it uses different control-flow graphs for computing down-safety and up-safety. Step 1601 (in FIG. 16) makes the computation of down-safety behave as if the tail of each (presumed) rarely taken edge is disconnected from the vertex that is connected to in the control-flow graph. Thus the results are based on the speculation that exceptions

are never thrown, and tend to cause the placement phase of the present invention to place instructions earlier (which tends to be optimistic) [[and]] than they would otherwise. Steps 1705-1706 (in FIG. 17) transfer the optimism to the up-safety calculation.